9. Opérateurs

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes
`=`	affect.	2	⟵	une var. et une val.
`+=`, `-=`, `*=`, `/=`, `%=`	affect. compo. arith.	2	⟵	une var. num. et une val. num
`&=`, `|=`, `^=`, `<<=`, `>>=`	affect. compo. bit à bit	2	⟵	une var. ent. et une val. ent.

Toute expression de la forme `a X= b` est équivalente à `a = a X b`.

Opérateurs d'affectation

Toutes les expressions d'affectation produisent une valeur qui est la **valeur qui vient d'être affectée**.

– Exemple –

Dans

```
int a, b;
a = 2;
b = 5;
a *= b += 3;
```

à cause de l'associativité des opérateurs d'affectation, la l. 4 s'interprète comme `a *= (b += 3);`.

Ainsi, comme `b += 3` produit la valeur `8`, `a` vaut finalement `16`.

9. Opérateurs

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes
`sizeof`	taille	1	–	~~une val.~~, ~~une var.~~ ou un **type**
`(T)`	coercition `T` est un type	1	–	une val.
`? :`	condition	3	–	une val. num. et deux val.
`,`	séquence	2	$\longrightarrow$	deux val.

L'opérateur de séquence

Dans l'expression `v1, v2`, où `v1` et `v2` sont des expressions, on commence par évaluer `v1` puis ensuite `v2`. Cette expression produit la valeur de `v1`.

Variante : avec des parenthèses, dans `(v1, v2)`, l'évaluation se fait aussi de la gauche vers la droite mais la valeur de l'expression est celle de `v2`.

L'opérateur `,` est le plus souvent utilisé dans les **champs des boucles** `for`.

– Exemple –

```
int i, j, l;
...
for (i = 0, j = l - 1 ; i < j ; ++i, --j) {
    ...
}
```

permet d'obtenir une boucle `for` avec **deux compteurs** : `i` croît et `j` décroît dans l'intervalle allant de `0` à `l - 1`.

10. Pointeurs de fonction

Plan

10. Pointeurs de fonction

Pointeurs de fonction

En C, il est possible de manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Cependant, au même titre qu'une variable, toute fonction possède une **adresse** en mémoire. Il devient alors possible de réaliser des opérations sur les fonctions au moyen de leur adresse.

On parle alors de pointeur de fonction.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

`&fct`

permet d'accéder à l'**adresse** de `fct`.

– Exemple –

```
#include <stdio.h>

int somme(int a, int b) {
    return a + b;
}

int produit(int a, int b) {
    return a * b;
}

int main() {
    printf("%p\n", &somme);
    printf("%p\n", &produit);
    return 0;
}
```

L'exécution de ce programme affiche `0x40052d` et `0x400541`. Ce sont respectivement les adresses des fonctions `somme` et `produit`.

Note : aux lignes 9 et 10, il est possible de ne pas mentionner les `&`. Le compilateur comprend implicitement qu'il s'agit de pointeurs de fonction.

Le type pointeur de fonction

Pour **déclarer** un pointeur de fonction, nous utilisons

```
T (*FCT)(T1, ..., TN);
```

où

- `T`, `T1`, …, `TN` sont des types ;
- `FCT` est un identificateur.

Ce pointeur de fonction possède `FCT` comme identificateur et est destiné à contenir l'adresse d'une fonction de type de retour `T` et qui possède `N` paramètres de types respectifs `T1`, …, `TN`.

– Exemple –

```
float moyenne(int a, int b) {
    return (0.0 + a + b) / 2;
}

/* Declaration d'un pointeur de fonction */
float (*moy)(int, int);

/* Utilisation */
moy = &moyenne;
printf("%f\n", moy(2, 3));
```

Pour la même raison que dans l'exemple précédent, il est possible à la ligne 9 de ne pas mentionner le `&`.

Cependant, pour la clarté du code, nous mentionnerons les `&` dès que nécessaire,

Champs pointeurs de fonction

Un **champ d'un type structuré** peut être un pointeur sur une fonction.

– Exemple –

Ceci déclare un type structuré censé modéliser des suites d'entiers :

```
typedef struct {
    int t;
    int (*t_suiv)(int);
} Suite;
```

```
int suivant(Suite *s) {
    s->t = s->t_suiv(s->t);
    return s->t;
}

int mul_2(int x) {
    return 2 * x;
}
```

Ici, `t` contient le terme courant de la suite et `t_suiv` est la fonction qui, étant donné un terme en entrée, calcule le terme suivant.

```
int main() {
    int i;
    Suite s;

    s.t = 1;
    s.t_suiv = &mul_2;
    for (i = 0 ; i < 6 ; ++i)
        printf("%d ", suivant(&s));
    return 0;
}
```

Ceci affiche `1 2 4 8 16 32`.

Tableaux de pointeurs de fonction

Il est possible de manipuler des **tableaux de pointeurs de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT)(T1, ..., TN);
```

2. on déclare ensuite le tableau de manière usuelle. Syntaxe :

```
FCT tab[M];
```

La syntaxe plus directe

```
T (*tab[M])(T1, ..., TN);
```

existe mais rend le code plus difficile à lire. Elle déclare un tableau `tab` de pointeurs de fonction sans la déclaration de type préalable de la 1re méthode.

Tableaux de pointeurs de fonction

– Exemple –

```c
/* Alias pour un type pointeur de fonction */
typedef int (*opb)(int, int);

int add(int a, int b) {
    return a + b;
}
int mul(int a, int b) {
    return a * b;
}
```

```c
int main() {
    /* Declaration du tableau de pointeur de fonctions */
    opb tab[2];

    tab[0] = &add;
    tab[1] = &mul;
    printf("%d %d\n", tab[0](10, 20), tab[1](10, 20));
    return 0;
}
```

Ceci affiche `30 200`.

Les tableaux de pointeurs de fonction peuvent être **dynamiques**.

– Exemple –

Dans l'exemple précédent, la ligne où le tableau est déclaré peut être remplacée par

```c
opb *tab;
tab = (opb *) malloc(sizeof(opb) * 2);
```

Plan

10. Pointeurs de fonction

Pointeur de fonction en paramètre

Une fonction peut être **paramétrée par un pointeur de fonction**. Un paramètre pointeur de fonction est spécifié avec la même syntaxe que celle qui sert à le déclarer.

– Exemple –

```
int appliquer(int n, int k, int (*f)(int)) {
    int i;
    for (i = 0 ; i < k ; ++i)
        n = f(n);
    return n;
}
```

est une fonction est paramétrée par un pointeur de fonction `f` acceptant un entier et renvoyant un entier.

– Exemples –

```
int add_1(int n) {
    return n + 1;
}
...
printf("%d\n", appliquer(3, 4, &add_1));
```

Calcule `(((3 + 1) + 1) + 1) + 1` et affiche `7`.

```
int mul_2(int n) {
    return 2 * n;
}
...
printf("%d\n", appliquer(3, 4, &mul_2));
```

Calcule `(((3 * 2) * 2) * 2) * 2` et affiche `48`.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le **type de retour** est un **pointeur de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias `R` pour le pointeur de fonction que l'on souhaite renvoyer ;
2. on définit la fonction souhaitée, dont le type de retour est `R`.

La **syntaxe plus directe**

```
R (*FCT(T1 ARG1, ..., TN ARGN))(R1, ..., RM) {
    /* Corps de la fonction */
}
```

permet de définir directement une fonction `FCT` de type de retour `R` et qui possède `M` paramètres de types respectifs `R1`, …, `RM`.

Cependant, le code devient illisible.

Renvoi d'un pointeur de fonction

– Exemple –

Ici, nous utilisons des pointeurs de fonction pour réaliser des opérations aléatoires.

```
/* Definition des operations */
int add(int a, int b) {
    return a + b;
}
int sub(int a, int b) {
    return a - b;
}
int mod(int a, int b) {
    return a % b;
}
```

```
/* Type de retour */
typedef int (*opb)(int, int);

opb op_alea() {
    opb tab[3];
    tab[0] = &add;
    tab[1] = &sub;
    tab[2] = &mod;
    return tab[rand() % 3];
}
```

Ceci affecte, de manière aléatoire et uniforme, la valeur `7`, `-1` ou `3` à `n`.

```
int n;
n = op_alea()(3, 4);
```

Plan

10. Pointeurs de fonction

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

– Exemples –

- Une fonction qui teste si deux valeurs sont égales ;
- une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite générique si elle peut représenter des données dont le type n'est pas fixé.

– Exemples –

- Une liste dont les éléments sont d'un type non fixé ;
- un arbre binaire dont les éléments sont d'un type non fixé ;
- un tableau dont les éléments sont d'un type non fixé.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est le type pointeur générique.

Pour convertir un pointeur générique `ptr_g` vers un pointeur d'un type connu `T`, on utilise l'**opérateur de coercition**

`(T *) ptr_g`.

Ceci est une expression de type `T`. Son évaluation ne provoque pas d'effet secondaire.

Avant de pouvoir interpréter (c.-à-d. déréférencer) la valeur située à une adresse spécifiée par un pointeur générique, **le convertir est primordial**.

Fonction générique de comparaison

Nous souhaitons comparer deux segments de la mémoire commençant respectivement à des adresses `x` et `y` s'étendant sur `nbo` octets.

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x; /* Coercition */
    yc = (char *) y; /* Coercition */
    for (i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i]) /* Dereferencement */
            return 0;
    return 1;
}
```

Pour effectuer cette comparaison, nous interprétons les zones de la mémoire à comparer comme des tableaux de `nbo` octets (type `char`).

Cette fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

```
ega(sizeof(T), &t1, &t2)
```

Ceci compare deux variables `t1` et `t2`, toutes deux d'un même type `T`.

– Exemple –

```
short tab1[8], tab2[8];
...
ega(sizeof(short) * 8, tab1, tab2);
```

Ceci compare le contenu des deux tableaux statiques `tab1` et `tab2`.

Fonction générique d'affichage de tableau

Nous souhaitons afficher un tableau `tab` de `n` cases de pointeurs génériques.

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;

    for (i = 0 ; i < n ; ++i) {
        aff_elt(tab[i]);
        printf(" ");
    }
}
```

Cette fonction est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction**.

Pour afficher un tableau générique, on procède en deux temps :

1. on implante **spécifiquement** une fonction `aff` de même type que `aff_elt` et qui permet d'afficher un élément du tableau ;
2. on appelle la fonction `aff_tab` avec cette fonction.

Fonction générique d'affichage de tableau

– Exemple –

```c
/* Fonction specifique d'affichage d'un element */
void aff_int(void *x) {
    int e;
    e = *((int *) x);
    printf("%d", e);
}

/* Version equivalente mais raccourcie */
void aff_int(void *x) {
    printf("%d", *((int *) x));
}

/* Utilisation */
aff_tab((void **) tab, 13, &aff_int);
```

Ceci affiche un tableau `tab` de taille `13` de pointeurs sur des **entiers**.

La coercition `(void **) tab` n'est pas obligatoire dans l'appel à `aff_tab`.

Fonction générique d'affichage de tableau

– Exemple –

```
typedef struct {
    int jour;
    int mois;
    int annee;
} Date;

/* Fonction specifique d'affichage d'un element */
void aff_date(void *d) {
    Date dd;
    dd = *((Date *) d);
    printf("%d-%d-%d", dd.jour, dd.mois, dd.annee);
}

/* Utilisation */
aff_tab((void **) tab, 23, &aff_date);
```

Ceci affiche un tableau `tab` de taille `23` de pointeurs sur des variables de **type structuré** `Date`.

La coercition `(void **) tab` n'est pas obligatoire dans l'appel à `aff_tab`.

Listes génériques

Nous souhaitons définir une structure de donnée **liste** dont le **type des éléments n'est pas fixé**.

```
typedef struct _Cellule {
    struct _Cellule *suiv;
    void *e;
} Cellule;

typedef Cellule *Liste;
```

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule.

Le type `Liste` permet ainsi de représenter des listes génériques.

C'est une structure de donnée générique car le **type des éléments** que les futures listes pourront contenir **n'est pas connu lors de l'écriture de la fonction**.

Listes génériques

Nous souhaitons afficher une liste générique `lst`.

```
void aff_lst(Liste lst, void (*aff_elt)(void *)) {
    Cellule *x;

    assert(lst != NULL);
    assert(aff_elt != NULL);

    for (x = lst ; x != NULL ; x = x->suiv) {
        aff_elt(x->e);
        printf(" ");
    }
}
```

Cette fonction est générique : elle permet l'**affichage** des éléments d'une liste générique.

Pour afficher une liste générique, on procède en deux temps (comme dans le cas des tableaux) :

1. on implante **spécifiquement** une fonction `aff` de même type que `aff_elt` et qui permet d'afficher un élément de la liste ;
2. on appelle la fonction `aff_lst` avec cette fonction.

– Exemple –

```c
/* Fonction specifique d'affichage d'un element */
void aff_int(void *e) {
    printf("%d", *((int *) e));
}

/* Creation d'une liste d'entiers */
Liste lst;
int a, b, c;

a = 3; b = 14; c = 414;
lst = (Cellule *) malloc(sizeof(Cellule));
lst->e = &a;
lst->suiv = (Cellule *) malloc(sizeof(Cellule));
lst->suiv->e = &b;
lst->suiv->suiv = (Cellule *) malloc(sizeof(Cellule));
lst->suiv->suiv->e = &c;
lst->suiv->suiv->suiv = NULL;

/* Utilisation */
aff_lst(lst, &aff_int);
```

Ceci créé une liste générique qui s'avère contenir des entiers.

Elle contient les éléments `3`, `14` et `414`.

L'appel à `aff_lst` affiche cette liste.

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques. Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`
4. `void *max(Liste lst, int (*est_inf)(void *, void *));`
5. `Liste empiler(Liste lst, void *e);`
6. `Liste miroir(Liste lst);`

Les cas 3 et 4 supposent que les éléments représentés par les listes sont comparables au moyen d'une fonction spécifique `est_inf` à fournir.

Plan

10. Pointeurs de fonction

Monoïdes

En maths, un monoïde est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. $\mathcal{M}$ est un ensemble ;
2. $\bullet$ est une application $\bullet : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ associative, c.-à-d., pour tous $x, y, z \in \mathcal{M}$, on a $(x \bullet y) \bullet z = x \bullet (y \bullet z)$;
3. $\mathbf{1} \in \mathcal{M}$ est un élément unitaire, c.-à-d., pour tout $x \in \mathcal{M}$, on a $x \bullet \mathbf{1} = x = \mathbf{1} \bullet x$.

– Exemples –

- $(\mathbb{N}, +, 0)$ est le monoïde additif des entiers naturels ;
- $(\mathbb{N}, \times, 1)$ est le monoïde multiplicatif des entiers naturels ;
- $(\mathbb{N}, \max, 0)$ est le monoïde max des entiers naturels ;
- $(\{\mathtt{a}, \mathtt{b}\}^*, \cdot, \epsilon)$ est le monoïde libre sur les lettres `a` et `b`. Ses éléments sont les chaînes de caractères composées de `a` et de `b`. Son produit $\cdot$ est la concaténation et son unité ϵ est la chaîne vide.